

Topic 8: Binary Trees¹

(Version of 12th December 2010)

Pierre Flener

Computing Science Division
Department of Information Technology
Uppsala University
Sweden

Course 1DL201:
Program Construction and Data Structures

¹Based on original slides by Yves Deville, and with pictures by John Morris and from the Wikipedia



Outline

Binary Trees

Binary
Search Trees

Balanced
Binary
Search Trees

- 1 **Binary Trees**
- 2 **Binary Search Trees**
- 3 **Balanced Binary Search Trees**



Outline

Binary Trees

Binary
Search Trees

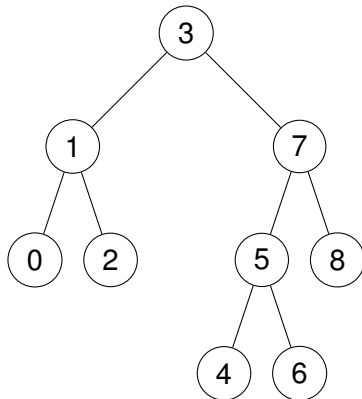
Balanced
Binary
Search Trees

- 1 **Binary Trees**
- 2 Binary Search Trees
- 3 Balanced Binary Search Trees



Binary Trees

Binary trees of elements of arbitrary type: 'a bTree



Terminology

- Node, root, leaf (plural: leaves), internal (inner) node
- Left and right subtree
- Parent, left and right child, sibling
- Edge, path, branch
- Level

Graphical Representation Convention

Empty trees are not drawn (but they consume memory).



Value and Some Operations

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

`empty`

TYPE: `'a bTree`

VALUE: the empty binary tree

`isEmpty T`

TYPE: `''a bTree -> bool`

PRE: (none)

POST: true if T is empty, and false otherwise

`cons (r, L, R)`

TYPE: `'a * 'a bTree * 'a bTree -> 'a bTree`

PRE: (none)

POST: the binary tree with root r, left subtree L,
and right subtree R



Some More Operations

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

left T

TYPE: 'a bTree -> 'a bTree

PRE: T is non-empty

POST: the left subtree of T

right T

TYPE: 'a bTree -> 'a bTree

PRE: T is non-empty

POST: the right subtree of T

root T

TYPE: 'a bTree -> 'a

PRE: T is non-empty

POST: the root of T



Representation and Implementation

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

```
datatype 'a bTree = Void
```

```
    | Bt of 'a * 'a bTree * 'a bTree
```

REPRESENTATION CONVENTION: the empty binary tree is represented by Void;

a binary tree with root r, left subtree L, and right subtree R
is represented by Bt(r,L,R)

REPRESENTATION INVARIANT: (none)

EX: Bt(4, Bt(2, Bt(1,Void,Void), Bt(3,Void,Void)),

Bt(8, Bt(6, Bt(5,Void,Void), Bt(7,Void,Void)), Bt(9,Void,Void)))

where bTree is a **type constructor**

while Void and Bt are **value constructors**.

```
val empty = Void
```

```
fun isEmpty T = (T = Void)
```

```
fun cons (r, L, R) = Bt(r,L,R)
```

```
fun left (Bt(r,L,R)) = L
```

```
fun right (Bt(r,L,R)) = R
```

```
fun root (Bt(r,L,R)) = r
```

Exercise: All these operations always take $\Theta(1)$ time.



Walks

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

Definition

A **walk** of a data structure is a way of listing each of its elements exactly once. For binary trees, we distinguish:

- the **preorder** walk (first list the **root**, then walk the left subtree, and last walk the right subtree)
- the **inorder** walk (left, **root**, right)
- the **postorder** walk (left, right, **root**)

Example

For the binary tree on **page 4**:

- preorder walk = 3 1 0 2 7 5 4 6 8
- inorder walk = 0 1 2 3 4 5 6 7 8 (coincidence?)
- postorder walk = 0 2 1 4 6 5 8 7 3



Inorder Walk

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

```

inorder  T
TYPE:  'a bTree          -> 'a list
PRE:   (none)
POST:  inorder walk of T

VARIANT: |T|
fun inorder  Void = []
  | inorder  (Bt(r,L,R)) =
    (inorder  L) @ (r :: inorder  R)

```

Double recursion, but **no** tail recursion.

Program uses $X@Y$, which takes $\Theta(|X|)$ time.

Exercise: `inorder T` takes:

- $\Theta(|T|)$ time at best (when L is always empty).
- $\Theta(|T| \cdot \lg |T|)$ time on average (when always $|L| = |R|$).
- $\Theta(|T|^2)$ time at worst (when R is always empty).



Generalisation by Accumulator Introduction

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

`inorder' (T, A)`

`TYPE: 'a bTree * 'a list -> 'a list`

`PRE: (none)`

`POST: (inorder walk of T) @ A`

`VARIANT: |T|`

`fun inorder' (Void, A) = A`

`| inorder' (Bt(r,L,R), A) =`

`inorder' (L, r :: inorder' (R, A))`

`fun inorder T = inorder' (T, [])`

Double recursion, but **one** tail recursion.

Program uses **no** $X@Y$, but the **specif.** of `inorder'` does.

Exercise: `inorder' (T, A)` and thus the new version of `inorder T` **always** take $\Theta(|T|)$ time.

Exercise: Implement efficient preorder and postorder walks of a binary tree, and analyse the designed algorithms.



Structural Generalisation

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

inorders Ts

TYPE: 'a bTree list -> 'a list

PRE: (none)

POST: (inorder walk of T1) @ ... @ (inorder walk of Tm),
 when Ts = [T1,...,Tm]

VARIANT: a tree is replaced by 0, 1, or 2 smaller trees

fun inorders [] = []

 | inorders (Void::Ts) = inorders Ts

 | inorders (Bt(r,Void,R)::Ts) = r::(inorders (R::Ts))

 | inorders (Bt(r,L,R)::Ts) =
 inorders (L::cons(r,Void,R)::Ts)

fun inorder T = inorders [T]

Single recursion: it is **even** a tail recursion in two clauses.

Exercise: For binary trees with a **total** of n nodes, inorders
and the new version of inorder **always** take $\Theta(n)$ **time**.



Structural Generalisation: Sample Trace

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

```

inorders [Bt(3,Bt(1,Void,Void),Bt(7,Void,Void))]
= inorders [Bt(1,Void,Void), Bt(3,Void,Bt(7,Void,Void))]
                                     by fourth clause
= 1 :: inorders [Void, Bt(3,Void,Bt(7,Void,Void))]
                                     by third clause
= 1 :: inorders [Bt(3,Void,Bt(7,Void,Void))]
                                     by second clause
= 1 :: 3 :: inorders [Bt(7,Void,Void)]
                                     by third clause
= 1 :: 3 :: 7 :: inorders [Void]
                                     by third clause
= 1 :: 3 :: 7 :: inorders []
                                     by second clause
= 1 :: 3 :: 7 :: []
                                     by first clause
= [1,3,7]
                                     by definition

```



Even More Operations

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

exists (T, k)

TYPE: 'a bTree * 'a -> bool

PRE: (none)

POST: true if T contains node k, and false otherwise

insert (T, k)

TYPE: 'a bTree * 'a -> 'a bTree

PRE: (none)

POST: T with node k

delete (T, k)

TYPE: 'a bTree * 'a -> 'a bTree

PRE: (none)

POST: if k exists in T, then T without **one** occurrence
of node k, otherwise T



Yet More Operations

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

`nbNodes T`

`TYPE: 'a bTree -> int`

`PRE: (none)`

`POST: the number of nodes of T`

`nbLeaves T`

`TYPE: 'a bTree -> int`

`PRE: (none)`

`POST: the number of leaves of T`

Exercises

- Implement efficient algorithms for these five functions.
- Show that these algorithms at worst take $\Theta(|T|)$ time, if not $\Theta(1)$ time.



Height

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

Definition

The **height of a node** is the length of the longest path (measured in its number of nodes) from that node to a leaf. The **height $h(T)$ of a tree T** is the height of the root of T .

Example: The binary tree on [page 4](#) has height 4.

```
height T
TYPE: 'a bTree -> int
PRE: (none)
POST: h(T)
VARIANT: h(T)    (note that |T| is also a variant)
fun height Void = 0
    | height (Bt(r,L,R)) = 1 + Int.max (height L, height R)
```

Double recursion, but **no** tail recursion.

Exercise: height T **always** takes $\Theta(|T|)$ time.



Generalisation by Accumulator Introduction

Note that $\text{height}'(T, a) = a + h(T)$
 does **not** suffice to get a tail recursion: Why?!

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

```
height' (T, a, hMax)
TYPE: 'a bTree * int * int -> int
PRE:  (none)
POST: max(a + h(T), hMax)
```

```
VARIANT: h(T)
fun height' (Void, a, hMax) = Int.max (a, hMax)
  | height' (Bt(r,L,R), a, hMax) =
    height' (L, a+1, height' (R, a+1, hMax))
fun height T = height' (T, 0, 0)
```

Double recursion, but **one** tail recursion.

Exercise: $\text{height}'(T, a, hMax)$ and thus the new version of `height T` **also** always take $\Theta(|T|)$ **time**, but much less **space** than the old version of `height T`.



Outline

Binary Trees

Binary
Search Trees

Balanced
Binary
Search Trees

- 1 Binary Trees
- 2 **Binary Search Trees**
- 3 Balanced Binary Search Trees



Binary Search Trees

Binary search trees of elements with integer keys and values of arbitrary type: `'a bsTree`

We **specialise** binary trees by a representation invariant:

REPRESENTATION INVARIANT: for a binary search tree with (k,v) in the root, left subtree L , and right subtree R :

- every element of L has a key smaller than k
- every element of R has a key larger than k

and recursively so on, for L and R

Example: The binary tree on page 4 is a binary search tree (whose values are not depicted).

Note that we (arbitrarily) ruled out duplicate keys.

Benefit: The inorder walk of a binary search tree lists its nodes by increasing order of their keys.

Question: Do we now have linear-time sorting?!



Representation and Implementation

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

```
datatype 'b bsTree = Void
```

```
    | Bst of (int * 'b) * 'b bsTree * 'b bsTree
```

REPRESENTATION CONVENTION: the empty binary search tree is represented by Void; a binary search tree with key-value pair (k,v) in the root, left subtree L, and right subtree R is represented by Bst((k,v),L,R)

REPRESENTATION INVARIANT: (see previous page)

```
empty
```

```
TYPE: 'b bsTree
```

```
VALUE: the empty binary search tree
```

```
val empty = Void
```

```
isEmpty T
```

```
TYPE: ''b bsTree -> bool
```

```
PRE: (none)
```

```
POST: true if T is empty, and false otherwise
```

```
TIME COMPLEXITY:  $\Theta(1)$  always
```

```
fun isEmpty T = (T = Void)
```



Existence Check

exists (T, k)

TYPE: 'b bsTree * int -> bool

PRE: (none)

POST: true if T contains a node with key k,
and false otherwise

VARIANT: $h(T)$

TIME COMPLEXITY: $\Theta(|T|)$ at worst (when $h(T)=|T|$)

fun exists (Void, k) = false

 | exists (Bst((key,value),L,R), k) =

 if k = key then true

 else if k < key then exists (L, k)

 else exists (R, k)



Search

Binary Trees

Binary Search Trees

Balanced Binary Search Trees

search (T, k)

TYPE: 'b bsTree * int -> 'b

PRE: k exists in T

POST: the value associated to key k in T

VARIANT: $h(T)$

TIME COMPLEXITY: $\Theta(|T|)$ at worst (when $h(T)=|T|$)

```
fun search (Bst((key,value),L,R), k) =  
    if k = key then value  
    else if k < key then search (L, k)  
    else search (R, k)
```



Insertion

Compare with the specification for binary trees on [page 13](#), so as to handle the assumed absence of duplicates in binary search trees:

```
insert (T, k, v)
```

```
TYPE: 'b bsTree * int * 'b -> 'b bsTree
```

```
PRE: (none)
```

```
POST: if k exists in T, then T with v as value for key k,  
      else T with node (k,v)
```

```
VARIANT: h(T)
```

```
TIME COMPLEXITY:  $\Theta(|T|)$  at worst (when  $h(T)=|T|$ )
```

```
fun insert (Void, k, v) = Bst((k,v),Void,Void)
```

```
  | insert (Bst((key,value),L,R), k, v) =
```

```
    if k = key then Bst((k,v),L,R)
```

```
    else if k < key then Bst((key,value), insert (L, k, v), R)
```

```
        else Bst((key,value), L, insert (R, k, v))
```



Deletion Issues

When deleting a node (key, value) whose subtrees L and R are **both** non-empty, we must not violate the representation invariant.

One option is:

- 1 Replace (key, value) by the node with the **maximal** key of L , as that key is **smaller** than the key of **any** node of R .
- 2 Remove this replacement node from L .

The other option is to replace (key, value) by the node with the **minimal** key of R , as that key is **larger** than the key of **any** node of L , and to remove that replacement node from R .



Help Function for Deletion

So we need a help function:

`extractMax T`

`TYPE: 'b bsTree -> (int * 'b) * 'b bsTree`

`PRE: T is non-empty`

`POST: (max, T'), where max is the node of T with
the maximal key, and T' is T without max`

`VARIANT: the number of elements larger than the root of T`

`TIME COMPLEXITY: $\Theta(|T|)$ at worst (when $h(T)=|T|$)`

`fun extractMax (Bst(r,L,Void)) = (r, L)`

`| extractMax (Bst(r,L,R)) =`

`let val (max, newR) = extractMax R`

`in (max, Bst(r,L,newR)) end`



Deletion

Binary Trees

Binary
Search Trees

Balanced
Binary
Search Trees

Compare with the specification for binary trees on **page 13**:

```
delete (T, k)
```

```
TYPE: 'b bsTree * int -> 'b bsTree
```

```
PRE: (none)
```

```
POST: if k exists in T, then T without the node with key k,  
      else T
```

ALGORITHM: when both subtrees of node of key k are non-empty, replace
that node by the node of maximal key in its left subtree

VARIANT: $h(T)$

TIME COMPLEXITY: $\Theta(|T|)$ at worst (when $h(T)=|T|$)

```
fun delete (Void, k) = Void
```

```
  | delete (Bst((key,value),L,R), k) =
```

```
    if k < key then Bst((key,value), delete (L, k), R)
```

```
    else if k > key then Bst((key,value), L, delete (R, k))
```

```
    else (* k = key *)
```

```
      case (L,R) of
```

```
        (Void,_) => R
```

```
        | (_,Void) => L
```

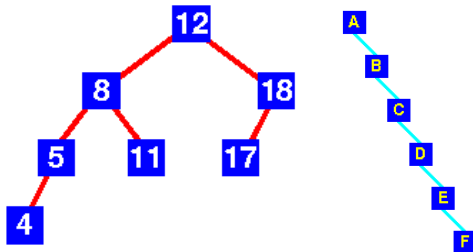
```
        | (_, _) => let val (max, newL) = extractMax L  
                     in Bst(max,newL,R) end
```



Observations

The search time in a binary search tree depends on the **shape** of the tree, that is on the order in which its elements were inserted.

The **A** pathological case: The n elements are inserted by increasing order on the keys, yielding something like a linear list (but with a worse space complexity), with $\Theta(n)$ search time at worst. Consider the tree on the right:





More Observations

Search, retrieval, insertion, and deletion take worst-case time proportional to the **height** of the binary search tree.

The height h of a binary tree of n elements is such that $\lg n < h \leq n$, so the four operations at worst take $\Theta(n)$ time.

The height of a **randomly built** (via insertions only, all keys being of equal probability) binary search tree of n elements is $\Theta(\lg n)$.

In practice, one can however not always guarantee that binary search trees are built randomly. Binary search trees are thus only interesting when they are “relatively complete.”

So we must look for a further **specialisation** of binary search trees, whose worst-case performance on the basic tree operations can be **guaranteed** to be logarithmic at most.



Outline

Binary Trees

Binary
Search Trees

Balanced
Binary
Search Trees

- 1 Binary Trees
- 2 Binary Search Trees
- 3 Balanced Binary Search Trees**



Balanced Binary Search Trees

“Definition”: A **balanced tree** is a tree where every leaf is “not more than a certain distance” away from the root than any other leaf.

The **balancing invariants** defining “not more than a certain distance” differ between various kinds of balanced trees:

- AVL trees (focus of this course)
- Red-black trees
- ...

Insertion and deletion involve transforming the tree if its balancing invariant is violated.

These re-balancing transformations must also take $\Theta(\lg n)$ time at worst, so that the effort is worth it. These transformations are built from operators (introduced on the next page) that are **independent** of the balancing invariant.

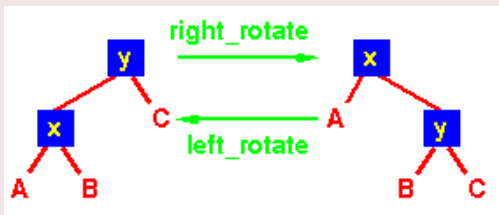


Rotations of Binary Trees

Definition

A **rotation** of a binary tree transforms the tree so that its inorder walk is preserved.

We distinguish **left rotation** and **right rotation**:



inorder walk = A x B y C

preorder walk = y x A B C

preorder walk = x A y B C

postorder walk = A B x C y

postorder walk = A B C y x

Exercise: Implement both rotations to take $\Theta(1)$ time.

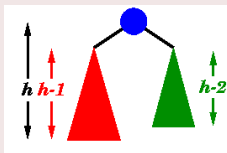


AVL Trees (Adel'son-Velskii & Landis, 1962)

AVL trees, the first **dynamically** balanced trees, are not perfectly balanced, but guarantee $\Theta(\lg n)$ worst-case search, insertion, deletion times for trees of initially n nodes.

Definition

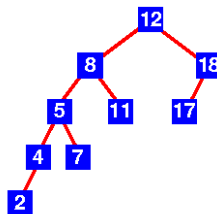
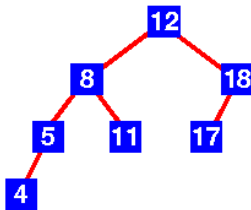
An **AVL tree** is a binary search tree with balancing invariant: The subtrees at every node differ in height by at most 1.



and conversely (when the left and right subtrees are exchanged) or when **both** subtrees are of height $h - 1$.



Two Examples and One Counter-Example



Let us annotate each node with a **balance factor** for the tree rooted at the considered node:

- if the tree is **stable** (when $r - \ell = 0$)
- if the tree is **left-heavy** (when $r - \ell = -1$)
- + if the tree is **right-heavy** (when $r - \ell = +1$)
- if the tree is **left-unbalanced** (when $r - \ell < -1$)
- ++ if the tree is **right-unbalanced** (when $r - \ell > +1$)

where ℓ and r are the heights of the left and right subtrees.



What Can We Expect from AVL Trees?

Key Questions

- What is the maximum height $hMax(n)$ of an AVL tree with n nodes?
- What is the minimum number $nMin(h)$ of nodes of an AVL tree of height h ?

Equivalent questions, but the second is easier to answer.

Recurrence:

$$nMin(h) = \begin{cases} 0 & \text{if } h = 0 \\ 1 & \text{if } h = 1 \\ 1 + nMin(h-1) + nMin(h-2) & \text{if } h > 1 \end{cases}$$



Search

Binary Trees

Binary
Search TreesBalanced
Binary
Search Trees

Compare the $nMin$ series with the Fibonacci series:

h	0	1	2	3	4	5	6	7	8	...
$nMin(h)$	0	1	2	4	7	12	20	33	54	...
$fib(h)$	0	1	1	2	3	5	8	13	21	...

Observe: $nMin(h) = fib(h + 2) - 1$. (**Exercise:** Prove this.)

Equivalently, the maximum height $hMax(n)$ of an AVL tree with n elements is the largest h such that:

$$fib(h + 2) - 1 \leq n$$

which simplifies into $hMax(n) \leq 1.44 \cdot \lg(n + 1) - 1.33$, so that search in an AVL tree takes $\Theta(\lg n)$ time at worst. Note that the `exists` and `search` algorithms for binary search trees work unchanged for AVL trees.



Insertion

How to insert — in logarithmic time — an element into an AVL tree such that it remains an AVL tree?

After locating the insertion place and performing a standard binary-search-tree insertion, there are only five cases:

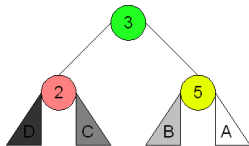
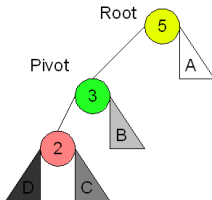
- 1 Every subtree remains balanced (\bullet , $-$, or $+$): done.
- 2 A left-heavy subtree became **left**-unbalanced ($--$):
 - a The balanced left subtree became **left**-heavy ($-$): right-rotate the left subtree toward the root.
 - b The balanced left subtree became **right**-heavy ($+$): first left-rotate the right subtree of the left subtree toward its parent, and then right-rotate the left subtree toward the root.
- 3 A right-heavy subtree became right-unbalanced ($++$): symmetric Cases 3a and 3b to Cases 2a and 2b above.



Insertion: Case 2a

Right-rotate left-heavy pivot 3 toward left-unbalanced root 5:

Left Left Case



Right
Rotation

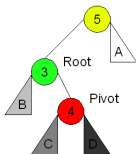
The inserted element is 2 (if C and D are empty) or a leaf of C or D. The trees A and B have height h , while the tree rooted at 2 had height h before the insertion and obtained height $h + 1$ after the insertion.



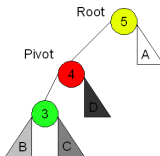
Insertion: Case 2b

First left-rotate the stable pivot 4 toward the right-heavy root 3, and then right-rotate the now left-heavy pivot 4 toward the still left-unbalanced root 5:

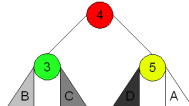
Left Right Case



Left
Rotation



Right
Rotation



The inserted element is 4 (if C and D are empty) or a leaf of C or D. The trees A and B have height h , while the tree rooted at 4 had height h before the insertion and obtained height $h + 1$ after the insertion.



Insertion and Deletion

Insertion Property

An insertion includes re-balancing

$$\Rightarrow (\nleftrightarrow)$$

that insertion does not modify the height of the tree.

Insertion: Insertion requires at most two walks of the path from the root to the added element, plus at most two constant-time rotations, hence insertion indeed takes $\Theta(\lg n)$ time at worst on an AVL tree of initially n nodes.

Deletion: The deletion of a node of given key from an AVL tree of initially n nodes can also be performed in $\Theta(\lg n)$ time at worst. (The algorithm is not studied in this course.)



When to Use Balanced Search Trees?

Dynamically balanced binary search trees are interesting when:

- The number n of elements is large (say $n \geq 50$),
and
- The keys are (suspected of) not appearing randomly,
and
- The ratio of the expected number s of searches to the expected number i of insertions is large enough (say $s/i \geq 5$) to justify the costs of dynamic re-balancing.